An overview and comparison of graph databases: Neo4j and AllegroGraph

Ruochen Deng, Wei-Hsiang Hung, Joycelyn Chen

April 23 2023

1 Introduction

1.1 What is a graph database?

A graph database, like any other database, stores data. The main difference between a graph database and the rest is that it stores nodes and relationships in a graph data structure instead of tables or documents like regular databases. These data are stored just like you might sketch ideas on a whiteboard, similar to a mind map. They can be stored without restricting them to a pre-defined model, allowing a very flexible way to think about and use them. [2]

Graph databases are purpose-built to store and navigate relationships. Graph databases use nodes to store data entities, and edges to store relationships between entities. An edge always has a start node, end node, type, and direction, and an edge can describe parent-child relationships, actions, ownership, and the like. There is no limit to the number and kind of relationships a node can have.

A graph in a graph database can be traversed along specific edge types or across the entire graph. In graph databases, traversing the joins or relationships is very fast because the relationships between nodes are not calculated at query times but are persisted in the database. Graph databases have advantages for use cases such as social networking, recommendation engines, and fraud detection, when you need to create relationships between data and quickly query these relationships.

We live in a connected world, it is only natural to connect the people, objects, events, or even relationships around us using a graph. Graphs have the advantage of displaying and having a comprehensive understanding of the whole picture. Often, we find that the connections between items are as important as the items themselves. Whether it's a social network, payment network, or road network, everything around us is an interconnected graph of relationships. And when we want to ask questions about the real world, many questions are about the relationships rather than about the individual data elements. [1]

Without graph databases, the existing relational databases store these relationships, they navigate them with expensive JOIN operations or cross-lookups, often tied to a rigid schema. These operations are often time-consuming and computationally expensive once the data amounts get large. However, in a graph database, there are no JOINs or lookups. Relationships are stored naturally alongside the data nodes in a flexible format. Everything about the system is optimized for traversing through data quickly.

In this project, we utilized Neo4j and AllegroGraph, two popular graph databases, to apply and reflect on the knowledge gained in our lecture. We conducted experiments by implementing graph traversal queries on both databases in order to compare their performance. Through rigorous testing and analysis, we evaluated the efficiency and effectiveness of the two databases in handling graph traversal queries, providing insights into their respective strengths and limitations. This allowed us to gain practical experience in working with graph databases and draw meaningful conclusions regarding their performance in real-world scenarios.

2 Backgorund

Currently, there is a lack of scholarly research that provides a comprehensive experimental analysis comparing different graph databases. In this study, our aim is to design and execute real-world queries

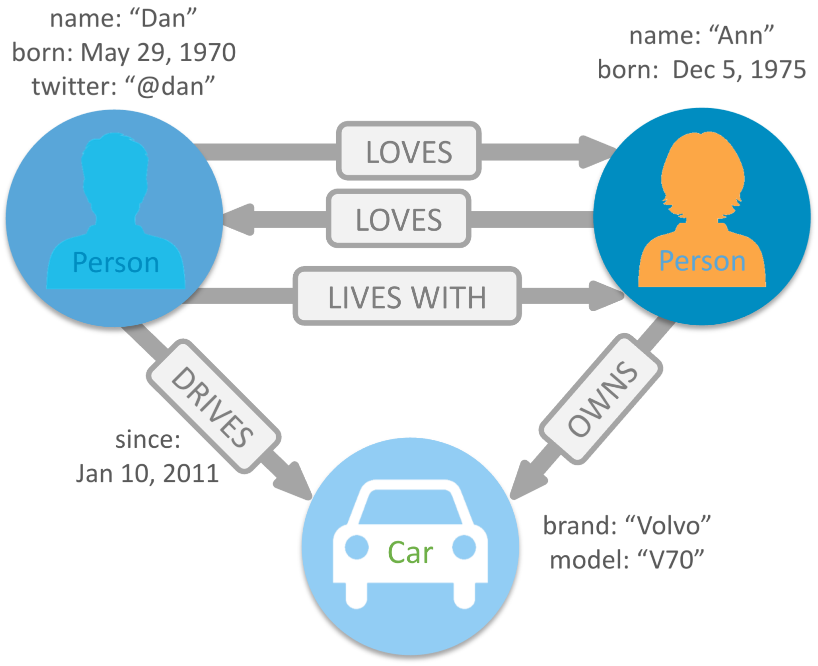

Figure 1: A tiny snippet of graph database demonstration. Each node represents a person or an object alongside the information attributes, while the edge connection between them represents the relationship. Source: Neo4j graph database [3]

and data structures in order to evaluate the performance of three competitive graph databases. The comparison will be based on four key objectives: database consistency, data loading efficiency, query performance, and data modeling expressive power. Subsequently, we will critically analyze and conclude the optimal choice for each scenario, and subsequently provide a summary of the characteristics, use cases, and limitations of the selected graph database.

3 Neo4j Database

Neo4j is the most well-known and widely used graph database, and is also a NOSQL system. When dealing with the large amount of real world network data or the highly interconnected data, Neo4j is an ideal choice to go for as it features the ability to store the data that have lots of connections between records with efficiency and scalability. Another feature is that Neo4j provides an ACID compliant transactional backend to maintain the data integrity within the database, which is extremely important as the connected data require a strict need for data integrity than any other NoSQL models. Last but not least, Neo4j developed a mature and intuitive query language, Cypher. The Cypher is inspired and similar to SQL, which made it friendly to graph database beginners. Assuming we have a IMDB dataset, we want to know the name of the user that has already watched the same movie as user 777, and we can retrieve the user name with $MATCH(u : USERid :' 777') - [: WATCHED] - (r)RETURNr$, where **WATCHED** is the relationship type, **user** is the node label, and **{id:'777'}** indicates the node with id 777. In the following paragraph, we will talk about the data storage, indexing, query processing, concurrency control, Recovery, and cluster in Neo4j.

3.1 Data storage in Neo4j

The data storage of the traditional tabular data is quite intuitive, however, storing a graph on disk sounds more counterintuitive. Neo4j is a native graph storage, indicating that it ensures the real world graph can be translated as primary, persistent data elements directly as opposed to imitating the graph functionality by transforming the graph into a traditional column or document paradigm, known as Non-native databases. Since native graph storage is built specifically for storing the graph, it usually performs queries faster and easier to scale.

As a native graph storage database, Neo4j is specifically designed to store graphs on disk, and it uses a proprietary data storage format called the property graph model to store the nodes, relationships and properties based on a set of files, allowing the data model of neo4j to map directly to how it works in the real world. Generally speaking, the storage engine is composed of *nodestore*, *relationshipstore*, *propertystore*, and *labelstore* file, and each is responsible for different graph information on disk. Neo4j is also a schema-less database, so they separate the field with offsets to locate the record under a fixed size record format. One of the key factors to distinguish good from bad graph databases lies in the capability to traverse the graph and answer the query in time. Because the storage engine in Neo4j is optimized for scalability and querying data by memory mapping and disk based storage, it is easier and more efficient to navigate the graph for the complex graph query.

3.2 Indexing in Neo4j

To further improve the performance of fetching the data, Neo4j provides two types of indexes, which are node indexes and relationship indexes and both operate in the same way. Neo4j offers four index types, including $LOOKUP, RANGE, POINT$, and $TEXT$, for different property predicates, while they removed the $BTREE$ index in 2020. The default index $LOOKUP$ is the most dominant type because it can be used to improve the population of other indexes and the query with node or relationship property, however, $RANGE$ indexes support equality check, existence check such as $IS\ NOT\ NULL$, prefix search like $START\ WITH$ and range search like *greater than*.

3.3 Query processing

In Neo4j, they use the cost-based query compiler, Ronja, to reach a superior execution time for Cypher queries under their extensive benchmarking tests. First, they convert the input query string into an abstract syntax tree (AST) and perform semantic checking within AST. Next, Ronja performs a simple optimization on AST before they create a query graph from it, and Ronja generates a logical plan for each query graph. Then, Ronja estimates the amount of work, which is the I/O reads and in-memory computational work, to get the cost of all logical plans. To pick the best logical plan, Ronja uses a greedy search strategy to select the cheapest logical plan as the query graph goes up. In the final step, Ronja rewrites the logical plan and picks the best physical implementation for the logical operators to output an execution plan.

3.4 Concurrency control

How does Neo4j keep the ACID compliant feature when multiple users run on their system concurrently? For two or more transactions, Neo4j has to make sure that the node or relationship is locked for a transaction so that any other transaction can not modify the node or relationship concurrently, which might break the data integrity. Neo4j implemented the optimistic locking, which is commonly used in relational databases. Instead of locking the data, which might impact performance, Optimistic locking allows the transaction to proceed without blocking, and checks for data conflict just before committing changes for the efficiency purpose as none of the transactions have to wait for the lock. The reason why this strategy would work is because it is less likely to have two transactions modifying the same data at the same time in a larger graph. Neo4j determines whether the data has been modified by a different transaction, leading to a timestamp differ, before it decides to commit or abort the transaction. In order to avoid the deadlock from happening, Neo4j utilizes the internally sorting operation to make sure the internal locks were obtained in order, but such strategy only applies when performing relations creation or deletion. If unfortunately it detects a deadlock, resulting in all of the transaction waiting for locking and none can proceed, the transaction will be terminated with an error message.

3.5 Recovery

Once a database becomes unavailable, Neo4j will employ its disaster recovery strategy. The first thing is to make the *system* database, which defines the configuration for other databases and is a vital role in Neo4j, fully operational to manage other databases. Neo4j follows the common strategies to recover the server, including backward scan over the log, transaction log replay to ensure that committed transactions are reflected, and rollback the incomplete transaction to keep the consistency. However, if the *system* has already lost too much data during the failover, Neo4j will follow the step to create a new *system* database from a backup. The next step is to recover other servers in the cluster with the same server recovery method, and then Neo4j identifies the lost servers and replaces them with a new server to update. When all the servers are up and running, it updates the status from unavailable to available.

3.6 Cluster

In practice, Cluster is a major topic for database systems, especially for commercial companies, as throughput, availability, and reliability of a database system is their top priority. The companies expect to answer as many users as possible and answer them at a speed of light with a minimum chance to failover and lose the data. A Neo4j cluster consists of servers in primary mode or secondary mode, and a single server is able to obtain two modes simultaneously. For example, a server can be a primary host for more than one database and acts as a secondary host for other databases at the same time. The cluster in Neo4j provides four main features, including safety, scale, causal consistency, and operability. Safety ensures the servers in primary mode are fault-tolerant and remains available, and scale feature makes sure the secondary mode servers provide a massively scalable for large workload queries, and the causal consistency guarantees that clients are able to read their own writes. Note that Neo4j only provides clustering in their enterprise edition. When adding a new member into the cluster, Neo4j needs to introduce the cluster to the new member by discovering the topology and providing the information such as the IP addresses and ports of at least some of the servers. Then, the server will send a handshake message to all the other servers based on the information so that the other clusters know the new members. Once the new member makes the connection to the primary node, it then officially joins the cluster.

Neo4j cluster is based on the RAFT protocol, which is a consensus algorithm to guarantee data consistency and availability in a group of interconnected servers. Neo4j follows the RAFT protocol to manage the server, record the membership change, keep the transaction log, and perform leader election. As a leader, it is responsible for replicating the transaction log to all the other nodes and waiting for the nodes to execute the log in its own database. If a majority of the nodes execute the transaction and reply an acknowledgement message such as a ”ready” message back to the leader, the leader will mark the transaction as ”Committed” because all the nodes now have the same data to answer the queries consistently. With the RAFT protocol, Neo4j is able to tolerate the node failure for database availability and ensure data consistency, presenting as a powerful distributed system.

4 AllegroGraph Database

AllegroGraph is a high-performance, scalable, and versatile graph database designed to handle complex, large-scale data. Developed by Franz Inc., it is an RDF (Resource Description Framework) database that employs the RDF triple store model and supports the SPARQL query language. AllegroGraph has gained recognition for its unique features and capabilities, making it an invaluable tool for complex data analysis tasks.

Graph databases have emerged as popular solutions for storing and analyzing complex data with multiple relationships. They use nodes and edges to represent entities and their relationships, as opposed to traditional relational databases that utilize tables and rows. Graph databases excel in scenarios where traditional databases struggle, such as social network analysis, recommendation systems, and fraud detection. They can model intricate relationships and provide insights that were previously difficult or impossible to obtain.

4.1 Resource description framework(RDF)

AllegroGraph, a RDF (Resource Description Framework) database that supports the SPARQL query language, utilizes the RDF triple store model. This model represents RDF data as triples, consisting of a subject, predicate, and object, allowing for flexible and extensible data interchange on the Web. With AllegroGraph, complex relationships and interconnections between entities can be efficiently stored, managed, and queried in the form of subject-predicate-object triples. Each triple represents a fact or statement about a resource, with the subject denoting the resource, the predicate representing a property or relationship, and the object indicating the value of the property or another related resource. This RDF triple store model empowers AllegroGraph to effectively model intricate relationships among subjects, predicates, and objects:

Subjects: In RDF, a subject is typically a resource identified by a URI (Uniform Resource Identifier), which serves as a unique identifier for the resource. In some cases, subjects can also be blank nodes, which are used to represent resources that do not have a URI.

Predicates: Predicates are used to describe the properties or relationships between subjects and objects. Like subjects, predicates are identified by URIs, ensuring that the meaning of the relationship is unambiguous and can be understood across different data sources.

Objects: Objects in RDF triples can be either resources or literals. Resources, like subjects, are identified by URIs, while literals represent simple values such as strings, numbers, or dates.

4.2 Querying RDF data with SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is the standard query language for RDF databases, allowing users to efficiently query and manipulate RDF data. It is designed to work with the RDF triple store model, providing a powerful and expressive syntax for querying complex relationships and patterns in RDF data.

There are several types of SPARQL queries, including:

- SELECT: Retrieves specific variables from the RDF data that match the specified patterns.
- CONSTRUCT: Creates new RDF triples based on the specified patterns and data.
- ASK: Returns a boolean value indicating whether the specified patterns exist in the RDF data.
- DESCRIBE: Retrieves an RDF description of the specified resources, including their properties and relationships.

4.3 Concurrency control

AllegroGraph employs several techniques to provide effective concurrency control:

1. Multi-Version Concurrency Control (MVCC): AllegroGraph uses MVCC to allow multiple transactions to operate concurrently without locking the entire database. MVCC creates multiple versions of the graph data for different transactions, allowing them to work independently without blocking each other. This enables concurrent read and write operations to be performed efficiently, reducing contention and improving performance.

2. Fine-Grained Locking: AllegroGraph uses fine-grained locking to ensure that only the necessary portions of the graph data are locked during concurrent transactions. Instead of locking the entire database or large portions of it, AllegroGraph employs locks at the level of individual RDF triples or small subsets of triples, minimizing contention and allowing concurrent operations to proceed with higher concurrency.

3. Locking and Logging: AllegroGraph uses locking mechanisms and logging to coordinate and manage concurrent access to the database. Locks are used to protect critical operations, such as updates to shared data structures, while logging is used to record changes made by transactions. These mechanisms help ensure that transactions are executed in a coordinated manner and that changes made by concurrent transactions are properly recorded and maintained.

AllegroGraph's concurrency control mechanisms are designed to provide efficient and effective management of concurrent access to the graph database, ensuring that transactions from multiple sources can be executed concurrently without interfering with each other.

4.4 Recovery

AllegroGraph, like other modern databases, employs various mechanisms to handle recovery in case of system failures or crashes. Recovery mechanisms are in place to ensure that the graph database can be restored to a consistent state after an unexpected event, such as hardware failure or power outage. Here are some of the key recovery mechanisms used in AllegroGraph:

1. Transaction Logging: AllegroGraph uses transaction logging to record changes made by transactions to the graph database. Transaction logs capture modifications to the graph data, such as inserts, updates, and deletions, as well as other metadata about the transactions. In case of a failure, the transaction log can be used to replay the changes made by transactions that were not yet persisted to the main database.

2. Crash Recovery: In case of a system crash or failure, AllegroGraph employs crash recovery mechanisms to restore the graph database to a consistent state. During the recovery process, AllegroGraph uses the transaction log and/or checkpoints to replay the changes made by transactions that were not yet persisted to the main database. This ensures that the graph database is restored to a consistent state and that no data is lost or corrupted.

3. Replication and High Availability: AllegroGraph supports replication and high availability configurations, where multiple copies of the graph database are maintained on different servers or clusters. In case of a failure in one server or cluster, the replicated copies can be used to continue serving the queries and maintaining data availability without interruption.

4.5 RDFS and OWL

One of AllegroGraph's key features is the support for RDFS (Resource Description Framework Schema) and OWL (Web Ontology Language) reasoning, which allows users to define and infer relationships between entities based on user-defined rules and ontologies. In this section, we will discuss how AllegroGraph leverages RDFS and OWL reasoning to deliver more intelligent and context-aware query results.

4.5.1 Resource description framework schema(RDFS)

RDFS is a semantic extension of RDF that provides a set of classes, properties, and constraints to define vocabularies and schemas for RDF data. RDFS reasoning in AllegroGraph allows for the inference of new RDF triples based on the existing data and the defined RDFS schema. There are some key concepts of RDFS:

Classes: RDFS classes represent categories or types of resources. Users can define their own classes using the rdfs:Class construct and then create instances of these classes using the rdf:type property.

Properties: Properties in RDFS represent relationships between resources or between resources and literals. Users can define their own properties using the rdf:Property construct and specify the domain and range of the property using rdfs:domain and rdfs:range, respectively.

Subclass and Subproperty: RDFS allows users to define hierarchies of classes and properties using the rdfs:subClassOf and rdfs:subPropertyOf constructs. This enables the inference of relationships based on class and property hierarchies.

Based on these concepts, AllegroGraph can perform RDFS reasoning on the RDF data, which includes inferring new triples based on the defined RDFS schema. Some examples of inferences that AllegroGraph can make using RDFS reasoning include: a)Inferring the rdf:type of a resource based on its class hierarchy, b)Inferring relationships between resources based on property hierarchies, c)Inferring the domain and range of properties based on their definitions.

4.5.2 Web ontology language

OWL is a more expressive ontology language that extends RDFS and provides additional constructs for defining complex relationships and constraints between resources. OWL reasoning in AllegroGraph allows for more advanced inferences based on the defined OWL ontology. The key concepts in OWL is different from RDFS:

OWL Classes and Individuals: OWL introduces a more expressive class system with constructs like owl:intersectionOf, owl:unionOf, and owl:complementOf for defining complex class expressions. Individuals are instances of OWL classes, and their relationships can be defined using OWL object and datatype properties.

Object and Datatype Properties: OWL differentiates between object properties (relationships between individuals) and datatype properties (relationships between individuals and literals).

Restrictions and Cardinality Constraints: OWL enables users to define restrictions on properties, such as the allowed values, the number of values, or the relationship between values.

AllegroGraph supports a subset of OWL reasoning insteads of whole power of it, focusing on features that can be efficiently implemented in a scalable graph database. Some examples of inferences that AllegroGraph can make using OWL reasoning include: a)Inferring relationships between individuals based on property restrictions and cardinality constraints. b)Inferring class membership of individuals based on complex class expressions and property restrictions. c)Inferring relationships between classes based on equivalence and disjointness axioms.

4.5.3 Benefits of RDFS and OWL reasoning in AllegroGraph

AllegroGraph's support for RDFS and OWL reasoning enables users to define and infer relationships between entities based on user-defined rules and ontologies. This powerful feature allows AllegroGraph to deliver more intelligent and context-aware query results, which can be particularly useful for complex data analysis tasks and applications. By harnessing the power of RDFS and OWL reasoning, AllegroGraph stands out as a efficient solution for managing and analyzing complex, large-scale graph data. Leveraging RDFS and OWL reasoning in AllegroGraph offers several benefits for users working with complex data:

- Enhanced Query Results: Reasoning enables AllegroGraph to deliver more intelligent and context-aware query results, as it can infer additional relationships and facts based on the defined rules and ontologies.
- Improved Data Consistency and Validation: RDFS and OWL schemas can be used to enforce data consistency and validation by specifying constraints and relationships between resources.
- Easier Data Integration and Interoperability: Using RDFS and OWL ontologies makes it easier to integrate and share data across different systems and applications, as it provides a standardized and unambiguous way to describe the structure and semantics of the data.

Table 1: The overview of our dataset.

	Squirrel	Crocodile	Chameleon
# of nodes	5201	11,631	2277
# of edges	217,073	180,020	36,101
graph density	0.0013	0.008	0.0069

Table 2: The query example for id 777.

	Query
1st order	Return the number of second order neighbors for id 777.
2nd order	Return the number of second order neighbors for id 777.
3rd order	Return the number of third order neighbors for id 777.
4th order	Return the number of fourth order neighbors for id 777.
5th order	Return the number of fifth order neighbors for id 777.

5 Experimental Result

Based on the previous sections, we had a comprehensive understanding of how the graph database worked in the real world, however, we were still curious about the graph traversal capability of different graph databases. As a result, we conducted the experiment with Neo4j and AllegroGraph database based on the WIKI dataset consisting of crocodile, chameleon, and squirrel dataset, shown in table 1.

The data utilized in this study was sourced from the English Wikipedia as of December 2018 [4]. The datasets consist of page-page networks that focus on specific topics, namely chameleons, crocodiles, and squirrels. In these networks, nodes represent articles and edges denote mutual links between them. The edges are stored in CSV files with node indices starting from 0, while the features are stored in JSON files, where each key represents a page ID and node features are presented as lists. The presence of a feature in the feature list signifies the appearance of an informative noun in the text of the corresponding Wikipedia article. Additionally, the target CSV file contains node identifiers along with the average monthly traffic data between October 2017 and November 2018 for each page. Descriptive statistics, including the number of nodes and edges, are provided for each page-page network to further characterize the datasets.

For each dataset, we queried the number of node of the first order neighbor until fifth order neighbor for id 777, 2041, and 319. The final execution time of was obtained by the average execution time of three ids. For example, we queried the number of third order neighbor for id 777, 2041, and 319 separately, and averaged the result of the number of records and execution time. The source code is available here.

5.1 Query performance

We prepared the neo4j.dump file beforehand by importing the nodes and edges csv file into the system and taking a snapshot for the system, which generates a .dump of the current state of the database with the Neo4j UI. The data loading time for the nodes and edges is presented in table 3, Note that the loading time here is calculated based on importing the original csv file not the time it takes to upload the prepared .dump file. With the dump file, the experimental result can be reproduced easily with the github repository. We prepared the .rdf file beforehead by importing the nodes and edges csv file into the system.

The result of Squirrel is shown in table 4. As a least dense graph, Chameleon dataset, presented in table 6, returned the least records of the fifth order neighbors with the fastest execution time, and the graph with the most relationships, which was the Crocodile dataset in table 5, took the longest to execute.

5.2 Comparison

The above tables compare the query performance of two graph databases, Neo4j and AllegroGraph, on three different datasets - Squirrel, Crocodile, and Chameleon. The tables show the execution time

Table 3: The data loading time in Neo4j.

	Squirrel	Crocodile	Chameleon
loading time(sec)/Nod4j	21	30	8
loading time(sec)/AllegroGraph	1.44	0.26	1.14

Table 4: The query performance result on Squirrel dataset. The result here is the average of id, 777, 2041, and 319. The 5th order neighbors of id 2041 takes more than hours so we set the time to ten times more than the maximum execution time of all queries, and so is the number of records.

	1st order	2nd order	3rd order	4th order	5th order
# of records	6	6	422	1026	1,614.5
execution time(ms)/Nod4j	1.7	1.5	235.9	27,801.5	347,761.1
execution time(ms)/AllegroGraph	42.6	42.5	50.2	2,327.2	1,128.7

in milliseconds for five different orders of neighbors (1st to 5th order), with the number of records increasing significantly with each order of neighbor.

5.2.1 Loading time comparison

Table 3 shows that AllegroGraph is significantly faster than Neo4j in terms of data loading time for all three datasets. The loading time for AllegroGraph ranges from 0.26 to 1.44 seconds, while for Neo4j, the loading time ranges from 8 to 30 seconds. We surmise that it's due to the differences in the data models and that they adopt different design principles and optimizations algorithm between the two databases. RDF is a flexible graph model that uses triples (subject-predicate-object statements) to represent data, and RDF databases are optimized for handling large-scale, distributed, and interconnected data. Neo4j, on the other hand, is a native graph database that uses a property graph model, where nodes represent entities and relationships between nodes are explicitly modeled as edges.

5.2.2 Query performance comparison

Overall, table 4 5 6 show that AllegroGraph outperforms Neo4j in terms of query performance, with faster execution times in most cases. Specifically, in the Squirrel dataset, AllegroGraph showed much faster performance for the 4th and 5th order neighbors. In the Crocodile dataset, AllegroGraph was faster for all orders of neighbors. In the Chameleon dataset, AllegroGraph was faster for the 2nd and 3rd order neighbors, while Neo4j was faster for the 4th and 5th order neighbors.

It is important to note that the number of records increases significantly with each order of neighbor, and for the 5th order neighbors, the execution time becomes very long, taking hours to complete. As a result, the table sets the time for the 5th order neighbors to ten times more than the maximum execution time of all queries, and similarly for the number of records. In conclusion, based on the results presented in the tables, AllegroGraph appears to be a more efficient graph database for handling large datasets with complex queries, as it consistently shows faster query performance than Neo4j. However, the choice of a graph database also depends on other factors such as ease of use, scalability, and cost, which should be taken into consideration when selecting a database for a specific use case.

Table 5: The query performance result on Crocodile dataset. The result here is the average of id, 777, 2041, and 319

	1st order	2nd order	3rd order	4th order	5th order
# of records	6	6	550	1364	2,212.6
execution time(ms)/Nod4j	1.2	3.8	73	11,366.3	769,562.5
execution time(ms)/AllegroGraph	16.8	22.9	28.0	178.6	5,241.4

Table 6: The query performance result on Chameleon dataset. The result here is the average of id, 777, 2041, and 319

	1st order	2nd order	3rd order	4th order	5th order
# of records	27	27	159	253	389
execution time(ms)/Nod4j	1.2	3.5	235.8	5208.8	712,470.8
execution time(ms)/AllegroGraph	18.1	36.8	69.6	3,261.8	5605.6

6 Conclusion

In conclusion, there is a need for more scholarly research that comprehensively compares different graph databases. In this study, our objective is to evaluate the performance of two competitive graph databases through real-world queries and data structures, focusing on key objectives such as data loading efficiency, and query performance.

When evaluating Neo4j and AllegroGraph databases for potential future applications, a comparison can be made in terms of concurrency control and recovery plans. Both databases utilize the multi-version concurrency control (MVCC) approach for managing concurrent access and provide mechanisms for recovery and backup/restore to ensure data consistency, integrity, and durability. However, there are differences in locking granularity, recovery mechanisms, and backup/restore options between the two databases:

1. Locking Granularity: Neo4j and AllegroGraph both use MVCC approach for concurrency control, but they differ in the granularity of locks used. Neo4j uses a combination of read and write locks at different granularities, while AllegroGraph uses fine-grained locking mechanisms, such as per-triple locks or predicate locks.

2. Recovery Mechanism: Both Neo4j and AllegroGraph use transaction logs (Neo4j) or write-ahead logs (AllegroGraph) to record changes made during transactions for recovery purposes. However, the specific implementation and performance of recovery differs in each database system.

3. Backup and Restore: Both Neo4j and AllegroGraph support backup and restore mechanisms for data durability and recovery. However, the specifics of these mechanisms, such as full and incremental backups, online or offline backups, and performance considerations vary between the two databases.

4. Concurrency Optimizations: Both Neo4j and AllegroGraph strive to optimize concurrency and minimize lock contention to allow for efficient concurrent access to the graph data. However, the specific optimizations and performance characteristics differs in each database system, depending on the underlying data model, storage, and indexing mechanisms.

Some of our group members did take the Special topic in Database course last semester, and according to her, some of the database terminology and materials was simply just a meaningless word then such as transaction. It was not until we got to the transaction part in this course did she fully understand what a the terminology is. In this project, we reflected on what we learn in the lecture with two famous graph database. After this course, we all become better, hopefully an expert, users and can understand most of the documents released by the database provider.

References

[1] Renzo Angles. A comparison of current graph database models. In *2012 IEEE 28th International Conference on Data Engineering Workshops*, pages 171–177. IEEE, 2012.

[2] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[3] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, volume 2324, 2013.

[4] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. Multi-scale attributed node embedding, 2019.